

MICROSOFT

October 15, 1981

Mr. Dennis M. Ritchie
Bell Laboratories
P.O. Box 304
Murray Hill, NJ 07874

Dear Mr. Ritchie:

We at Microsoft are involved in transporting UNIX, under the name XENIX, to a variety of microprocessors, including the Z8000, M68000, and Intel 8086. As a result, we are intimately involved with the portability and consistency of the C programming language. I have found that there is a great deal of confusion and inconsistency in the handling of type casts and sign/zero extension. In view of the proliferation of micros running UNIX as well as the existence of several independent compilers for each of these micros, I feel that it is important that a clear and definitive standard be established.

Of course, the most critical test of any standard is that it be accepted. This is why I'm asking you, as the foremost authority on the "C" language, to endorse a standard. I naturally have my own opinions on the subject (enclosed), but more important than the form of the standard is the fact that there IS a standard.

It is my intention to present a paper at the January USENIX conference describing the "official, no-questions-about-it standard". I will also describe how current compilers depart from this standard, and present suggestions for coding styles that produce identical results under both "standardized" and existing compilers.

Thank you in advance for your input.

Respectfully,

Hans Spiller
XENIX Group

HS/jr
Enclosure

HANS SPILLER
"C" LANGUAGE

October 15, 1981

The area that has come to my attention is that of sign versus zero extension in the various compilers, both in the context of expressions and type casts. It seems to me that the existing definition is at best inconsistent and unclear (C reference manual 6.1, 6.5, and 6.6).

As I read it, when converting from char to any longer type, the char is extended to the size of int before any other extension or operation. The type of extension, signed or unsigned, depends upon the machine for type 'char', and is unsigned for type 'unsigned char'. Then, should further extension be necessary, and either operand be unsigned, zero extension is done. It seems to me that by this definition the expression:

```
longvar = (unsigned)charvar;
```

should sign extend the char to the size of an int, and then zero extend that to the size of a long before doing the assignment. In fact, the pcc sign extends the char all the way to long, and your version 7 compiler zero extends the whole thing.

It seems to be that a much better definition would be to simply say:

"When converting a shorter type to a longer, sign extension is done if both types are signed. Otherwise zero extension is done."

It happens that this is consistent, as far as I have been able to determine, with your PDP-11 compiler, except in the case of assignments, in which case you sign extend. For example,

```
unsigned int u;  
char c;  
u = c;
```

sign extends the signed character 'c' to an unsigned int (which I think is wrong), whereas

```
u = (unsigned)c;
```

zero extends (which I think is right). The pcc bases sign extension fairly strictly on the smaller type, regardless of casts, so both examples will sign extend. This seems wrong to me. If a user says he wants conversion to unsigned, sign extension is an incorrect operation, as part of the definition of an unsigned number is that there is no sign.

Please tell me what you think. I will be distributing compilers based on the resolution of this issue to many thousand customers. The fact that all the compilers implement it differently (We bought a Z8000 compiler based on the pcc that has yet a different interpretation.) raises serious portability problems. I have modified our z8000 compiler to implement my interpretation; it was a very trivial modification to the tables for the conversion operator. It should be equally trivial for any other compiler to be consistent with this approach.



Bell Laboratories

600 Mountain Avenue
Murray Hill, New Jersey 07974
Phone (201) 582-3000

October 23, 1931

Mr. Hans Spiller
Microsoft
10300 NE Eighth, Suite 319
Bellevue,
Washington 98004

Dear Mr. Spiller:

You are right that the published C manual is somewhat unclear, and existing compilers inconsistent, about type casts and sign extension. There has been some development in this area since the 7th edition system was released. Here is the current situation.

The key idea for casts is that they should behave as if the subject expression were assigned to a temporary variable of the type of the cast, and then the temporary used in place of the cast expression. In this way there is only one set of rules, as expressed in section 6.6.

Here is how I would apply the rules to your examples. First, "longvar = (unsigned)charvar": sign extend to the size of an int; then convert to unsigned; then zero-extend this quantity to long. In this way (on the PDP-11) the character 0377 would become 0177777, and then 0177777L. Similarly, "u = c" sign extends.

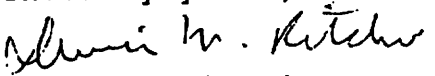
Now this is what our current compiler does, and what the V7 pcc compiler did, and it is also what you find odd. Actually it is strange, and the reason is that what is being asked is strange in itself. Given that characters are signed, the value of the character 0377 is in fact -1. There is just no way in which -1 can be converted to an unsigned number without anomaly. The problem with earlier PDP-11 compilers is that they did not implement the type "unsigned char," and the special properties of the "(unsigned)charvar" cast were an attempt to find a cheap way around this.

The point is that if one wants to store 0377 in an 8-bit character, and to have that value be 255 instead of -1, the character variable itself had better be unsigned. I would like to move towards a language in which the type declarations describe the range of values that can be stored

in a variable, rather than getting involved in questions of sign-extension, which is a very representation-dependent notion. From this point of view, applying "(unsigned)" to a negative number, whether stored in 3 or 16 bits, is simply an error (or at least non-portable). This view of things can be carried out pretty well if the type "unsigned char" is available.

If you wish to discuss this further, please feel free to write again, to telephone me at (201) 582-3770, or send computer mail to ucbvax!research!dmr (uucp) or csvax.dmr@berkeley (Arpanet).

Sincerely yours,


Dennis M. Ritchie

Sign Extension and Portability in C

Hans Spiller

XENIX Group
Microsoft, Inc.
Bellevue, WA

ABSTRACT

The definition of the C language has left confusion regarding type casts, sign extension, and the char type. This paper discusses the Bell standard and describes non-compatible and even incorrect code in existing compilers. Due to the prevalence of such compilers, macros are presented which will convert consistently, even given an incorrect compiler. A change to the type conversion rules in The C Programming Language is proposed.

The Issue

C has 3 integral sizes: char, short, and long. These integers can, in principle, be signed or unsigned. However, early implementations did not implement unsigned, and not all current implementations implement unsigned for all sizes. Some early implementations also did not support long.

C supports both automatic and explicit conversions between any of these types. (And a few others, for that matter...) In general, converting from a longer size to a shorter is easy: just forget about the high order bits and you're done. However, in converting from a shorter to a longer type you need some way of deciding what the high order bits will be. Normally, zero extension or sign extension are used. Precisely which extension is chosen is often relevant.

```
char c;  
int i;
```

- 1) if (c==0200)
- 2) i |= c;

In the first example - the comparison - the constant 0200 is an int, and has leading zeros on. To do the comparison, the char is sign extended to int; and should it have started out with the value 0200, it ends up with the value 0177600, and the test fails. Thus the test always fails. A good optimizing compiler could remove the test and whatever code might be executed should the test be successful.

In the second example, if the sign bit (bit 7) of the char is on, all the high order bits of the result are set regardless of their previous value. This could be a surprise to an unsuspecting user. However, if zero extension is done, the high order bits are left alone.

Modern C compilers provide a special type to do this, unsigned char, which has the property that zero extension rather than sign extension is done on instances of it. They also provide a mechanism (inline type casts) whereby you can specify unsignedness within an expression should you have the wrong type to start out with. There are unfortunately several problems.

The Problem

- 1) Not all C compilers support declarations or casts of all unsigned types.
- 2) Not all C compilers, even when they provide these types, use the same algorithm for choosing the conversions.
- 3) The defining document The C Programming Language, by Kernighan and Ritchie, known as the white book, explains the the standard in a confusing and almost ambiguous way. Worse, it says that whether the type char is signed or unsigned is machine dependent.
- 4) Given that an implementation takes advantage of the looseness of the white book and implements char as unsigned, there is no way to do sign extension of a char within the language.

If we are interested in portability, saying that something is to be machine dependent is useless. Worse yet, that the type of extension done is machine dependent is not even true. I have examples from three compilers for the PDP-11 below, and they are all different. And even though the compilers are allowed to differ where it is hard for the hardware to sign extend, most of the compilers for other machines, including Microsoft's 28000, 8086, and 68000 compilers, all try to simulate the PDP-11 by sign extending, even when it is painful.

The reasons behind all of this are primarily historical, but the big motivating factor is trying to provide something useful. Unfortunately, before the full generality of some things had been thought out, their usefulness had gotten to the point that they were implemented.

Whats Out There

It turns out that Bell has put forward, in the form of implementations, a standard, even though many of its own compilers do not conform to it. That this Bell's intent has been confirmed by Dennis Ritchie, in a letter he wrote me last October 23. Loosely, the standard is as described in the white book, with the principle difference being that char is always signed. What the white book says about when to sign extend is in fact consistent, but the way it accomplishes this is different based upon how you read the rules. The rules are not actually ambiguous, but they are tricky. And they are by no means easy to follow. What Ritchie, the current Bell compilers, and the white book all say is that you sign extend when the operand is signed, and you zero extend only when the source is unsigned.

Not all the recent compilers from Bell implement this rule. Conveniently, there are two different families of compilers for C available on the PDP-11, one written by Dennis Ritchie of Bell Labs, which I call the Ritchie compiler, and one written by Steve Johnson, also of Bell Labs, which I call the Portable C Compiler, or PCC. I have considered two different implementations of the Ritchie compiler, one from version 7, and one from the new release 3.0. The test program is basically the same for all three, except where the Version 7 Ritchie compiler objects. For this example, the 3.0 PCC does exactly the same thing as the Version 7 PCC:

```
int i;
char c;
unsigned int ui;
unsigned char uc;
main()
{
    i = c;
    ui = c;
    i = uc;
    ui = uc;

    i = (unsigned)c;
    ui = (unsigned)c;
    i = (unsigned)uc;
    ui = (unsigned)uc;

    i = (unsigned char)c;
    ui = (unsigned char)c;
```

January 22, 1982

```

        i = (unsigned char)uc;
        ui = (unsigned char)uc;
    }

```

V7 PCC	V7 Ritchie	3.0 Ritchie

First, uncasted assignments from 8 to 16 bits.

<pre> /i=c; movb c,r0 mov r0,_i /ui=c; movb c,r0 mov r0,_ui /i=uc; movb uc,i bic \$!377,_i </pre>	<pre> /i=c; movb c,r0 mov r0,_i /ui=c; movb c,r0 mov r0,_ui </pre>	<pre> /i=c; movb c,r0 mov r0,_i /ui=c; movb c,r0 mov r0,_ui /i=uc; clr r0 bisb _uc,r0 mov r0,_i /ui=uc; clr r0 bisb _uc,r0 mov r0,_ui </pre>
--	--	--

/not implemented

Up to this point, all three compilers agree, as far as they go. The specific code generated is different, but it does the same thing. But notice that the V7 Ritchie compiler does not implement unsigned char.

Now cast to unsigned:

V7 PCC	V7 Ritchie	3.0 Ritchie

<pre> /i=(unsigned)c; movb c,r0 mov r0,_i </pre>	<pre> /i=(unsigned)c; movb c,r0 bic \$-400,r0 mov r0,_i </pre>	<pre> /i=(unsigned)c; movb c,r0 mov r0,_i </pre>
<pre> /ui=(unsigned)c; movb c,r0 mov r0,_ui </pre>	<pre> /ui=(unsigned)c; movb c,r0 bic \$-400,r0 mov r0,_ui </pre>	<pre> /ui=(unsigned)c; movb c,r0 mov r0,_ui </pre>
<pre> /i=(unsigned)uc; movb uc,i bic \$!377,_i </pre>	<pre> </pre>	<pre> /i=(unsigned)uc; clr r0 bisb _uc,r0 mov r0,_i </pre>
<pre> /ui=(unsigned)uc; movb uc,ui bic \$!377,_ui </pre>	<pre> </pre>	<pre> /ui=(unsigned)uc; clr r0 bisb _uc,r0 mov r0,_ui </pre>

/not implemented

Up to this point, the 3.0 Ritchie compiler and the PCC are in agreement. The type of extension is based on the type of the operand to the cast operator, but the cast itself has no apparent effect on the generated code, apparently because it

is the same size as the destination of the assignment. The version 7 Ritchie compiler does something quite different, however. Rather than considering the type of the operand in determining the extension, the cast is used. This has the advantage that users have the opportunity of determining the extension they want, even though the full generality of the white book and 3.0 compiler is not implemented.

Now cast to unsigned char:

V7 PCC	V7 Ritchie	3.0 Ritchie

/i=(unsigned char)c;	/not implemented	/i=(unsigned char)c;
movb c,r0		movb c,r0
mov r0,_i		bic \$-400,r0
		mov r0,_i
/ui=(unsigned char)c;		/ui=(unsigned char)c;
movb c,r0		movb c,r0
mov r0,_ui		bic \$-400,r0
		mov r0,_ui
/i=(unsigned char)uc;		/i=(unsigned char)uc;
movb uc,i		clr r0
bic \$!377,_i		bisb uc,r0
		bic \$-400,r0
		mov r0,_i
/ui=(unsigned char)uc;		/ui=(unsigned char)uc;
movb uc,ui		clr r0
bic \$!377,_ui		bisb uc,r0
		bic \$-400,r0
		mov r0,_ui

Here, the V7 Ritchie compiler does not even try, because with casts, whatever conversion is wanted can be generated, but the PCC does something which is useless. I'm fairly sure its a bug, but it completely ignores the cast. Remember that casts are supposed to be treated as though there were an assignment to a variable of the type of the cast? Where'd it go? In general, it would appear that the V7 PCC bases sign extension strictly on the type of the source operand, and ignores casts and the destination type, while the Ritchie compiler bases sign extension on the source type, unless casts are present, in which case it bases it upon the type of the cast. This means that with the PCC on the 11, it isn't possible to do unsigned extension without explicitly putting the interceding assignment in, or by masking, should the source be in a signed type. It turns out that this problem persists when converting an int to a long in the PCC.

Despite the fact that the Ritchie compiler does not provide any way to do anything with unsigned chars, it does provide what is necessary: if you have an 8 bit quantity, you can decide with the cast operator whether you want it to be sign extended or not in a fairly convenient way. This is

useful, and seems to me to be quite natural. Unfortunately, the PCC did not implement casting this way. It continues to use only the type of the source to determine the type of the cast, so there is no way to write code that will do the same thing under both version 7 compilers without explicitly masking.

The Microsoft compilers for the Z8000 and 68000 implement the same rules as the 3.0 Ritchie for these examples. The current Microsoft 8086 compiler implements the same rules as the V7 Ritchie compiler. (Little guys have these problems too...)

I have another example that I think is interesting:

```
char c;
long l;
main()
{
    l = (unsigned)c;
}
```

V7 PCC	V7 Ritchie	3.0 Ritchie

movb c,r1	movb c,r0	movb c,r0
mov r1,_l+2.	bic \$-400,r0	mov r0,2+_l
sxt _l	mov r0,2+_l	clr _l
	clr _l	

The V7 and 3.0 PCC sign extend this, the V7 Ritchie zero extends, and the 3.0 Ritchie sign extends to short, and then zero extends to long. If you use the current Bell standard - using the type of the operand of each operator to determine extension and applying it first to the signed char source, sign extending it to unsigned, and then zero extending the unsigned to signed long - then only the 3.0 Ritchie compiler produces correct code.

Dennis Ritchie, in personal correspondence, agrees that this mixed sign/zero extension is odd, but points out that what is being done is odd in itself, in that a signed quantity is being forced to fit into an unsigned hole. Extending this argument, it seems to me that as soon as you know that there is something odd going on with signedness, all the compiler can possibly know about what is happening is that there is bit pushing going on that happens to have once been based on signed numbers.

But that doesn't resolve the problem, except in the best of all possible worlds, where everybody has compilers that work consistently for a variety of machines. Bell seems to be going in that direction with release 3.0, but they are not there yet with the PCC as of 3.0, and there are

all sorts of strange variant compilers in the world, including some that Bell itself has produced.

Because the early compilers only provided sign extension, and the unsigned data type was only gradually hacked in, there has been a great deal of inconsistency in the implementation of the unsigned data type and the cast. In order to make C a more portable language something must be done.

What Can Be Done

The principle problem that must be dealt with is that many compilers don't support, or have limited support for, unsigned data types. In general it is easy to get the compiler to sign extend because that is the historical antecedent: all the compilers I have looked at, sign extend by default when converting a char to an int. I do know of a number of exceptions. But because of the inconsistency in the various compilers, if you want to zero extend, you had best do it yourself. This can be done by bitwise ANDing the char with 0377 to convert it to a short, or ANDing an int with 017777 to convert it to a long.

The big advantage to doing this rather than trying to rely on consistent compilers in the future is that you can be confident that it will always work, even if you are using one of the very old compilers that doesn't have unsigned at all. In fact, much of the UNIX* system and its utilities are written this way. It may not generate quite as good code in some compilers, but at least you are portable.

The problem remains with the perverse compilers that do not do sign extension. I have only painful things to say to the user of such a compiler who needs sign extension.

- 1) I think the compiler you are using should not be called a C compiler. Thus Honeywell-IBM users are in trouble.
- 2) You really should fix the compiler, or try to get it fixed.
- 3) You are going to have to write code to do the sign extension explicitly:

```
#define CTOS(x) ((x&0200)?(x|0177400):(x&0377))  
#define STOL(x) ((x&0100000)?(x|037777600000L):(x&0177777L))
```

*UNIX is a Trademark of Bell Laboratories.

January 22, 1982

Just for completeness here is the code to zero extend,
as I described it above:

```
#define ZCTOS(x) (x&0377)
#define ZSTOL(x) (x&0177777L)
```

What I would like to see done

The solution is twofold:

- 1) Fix all the compilers for all the machines so they are completely consistent.
- 2) Fix the white book so it clearly defines exactly at what points sign extension and zero extension are done, and change the definition so that there is no machine dependency.

I am making all the Microsoft C compilers obey the Bell standard. They mostly do already, much more so than either of the V7 compilers. I have come up with a proposed new definition that I think is clearer and more easily applied than what is currently in the white book. It would involve altering Appendix A, The C Reference Manual.

Appendix A, Section 6.1 should be rewritten to say that chars are signed, unsigned chars are unsigned.

The relevant sections of Appendix A, 6.5 and 6.6 should be replaced by the following:

- 0) Definitions
An operation may be either arithmetic or logical (bit-wise)
An integer may be either signed or unsigned.
- 1) The types char, int, short, and long are signed.
The types unsigned char, unsigned int, unsigned short, unsigned long, and unsigned are unsigned. A pointer to anything is unsigned.
- 2) No integer operation is done in a type shorter than the length of int. No integer operation is done in a type shorter than the longest operand.

January 22, 1982

- 3) When combining an unsigned and a signed integer in an operation, the result is unsigned.
- 4) When converting a shorter type to a longer type, if the source type is unsigned, the source is zero padded to the length of the longer type. If the source type is signed, it is sign extended.
- 5) When converting a longer type to a shorter type, high order bits are ignored, and the low order bits are considered to be of the shorter type.
- 6) It is not necessary that all the conversions actually take place. It is only necessary that results be as if all conversions had taken place.

Note that I don't address floating point here. Even though the white book says that floating point tends to be fairly machine dependent, (and believe me, it is) for some reason it hasn't been much of a problem. I suspect this is because the problems are mostly with loss of precision, rather than blatantly wrong results. I also suspect most people that really care about floating point use FORTRAN.

The other issue I don't address is machines which encourage character sets that take up 8 bits, and have useful characters with the high order bit set. CDC and IBM have such machines. It seems to me the only option open is to preserve the rules above, and to drop the restriction that chars have to be in the positive range of the character. Then we would have to insist that users playing the usual char-'0' tricks declare their characters unsigned if they want to be portable. These tricks don't work in EBCDIC anyways. I know very little about the use of C on such machines. Perhaps it is unimportant.